BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
09/703,356	10/31/2000	Stepan Sokolov	SUN1P810/P5510	3230

22434 7590 07/13/2004

BEYER WEAVER & THOMAS LLP
P.O. BOX 778
BERKELEY, CA 94704-0778

EXAMINER	
OPIE, GEORGE L	

ART UNIT	PAPER NUMBER
2126	

DATE MAILED: 07/13/2004

Please find below and/or attached an Office communication concerning this application or proceeding.

Office Action Summary	Application No.	Applicant(s)
	09/703,356	Sokolov et al.
	Examiner	Art Unit
	George L. Opie	2151

-- The MAILING DATE of this communication appears on the cover sheet with the correspondence address --

Period for Reply

A SHORTENED STATUTORY PERIOD FOR REPLY IS SET TO EXPIRE 3 MONTH(S) FROM THE MAILING DATE OF THIS COMMUNICATION.

- Extensions of time may be available under the provisions of 37 CFR 1.136 (a). In no event, however, may a reply be timely filed after SIX (6) MONTHS from the mailing date of this communication.
- If the period for reply specified above is less than thirty (30) days, a reply within the statutory minimum of thirty (30) days will be considered timely.
- If NO period for reply is specified above, the maximum statutory period will apply and will expire SIX (6) MONTHS from the mailing date of this communication.
- Failure to reply within the set or extended period for reply will, by statute, cause the application to become ABANDONED (35 U.S.C. § 133).

Status

1) ___ Responsive to communication(s) filed on _____.

2a) ___ This action is **FINAL**. 2b) ___ This action is non-final.

3) ___ Since this application is in condition for allowance except for formal matters, prosecution as to the merits is closed in accordance with the practice under *Ex parte Quayle*, 1935 C.D. 11, 453 O.G. 213.

Disposition of Claims

4) X Claim(s) 1-15 is/are pending in the application.

4a) Of the above claim(s) _____ is/are withdrawn from consideration.

5) ___ Claim(s) _____ is/are allowed.

6) X Claim(s) 1-15 is/are rejected.

7) ___ Claim(s) _____ is/are objected to.

8) ___ Claim(s) _____ are subject to restriction and/or election requirement.

Application Papers

9) ___ The specification is objected to by the Examiner.

10) ___ The drawing(s) filed on _____ is/are objected to by the Examiner.

11) ___ The proposed drawing correction filed on _____ is: a) ___ approved b) ___ disapproved.

12) ___ The oath or declaration is objected to by the Examiner.

Priority under 35 U.S.C. § 119

13)_ Acknowledgment is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d).

a) ___ All b) ___ Some * c) ___ None of the CERTIFIED copies of the priority documents have been:

1. ___ received.
2. ___ received in Application No. (Series Code / Serial Number) _____.
3. ___ received in this National Stage application from the International Bureau (PCT Rule 17.2(a)).

* See the attached detailed Office action for a list of the certified copies not received.

14) ___ Acknowledgement is made of a claim for domestic priority under 35 U.S.C. & 119(e).

Attachment(s)

14) X Notice of References Cited (PTO-892)
15) ___ Notice of Draftsperson's Patent Drawing Review (PTO-948)
16) X Information Disclosure Statement(s) (PTO-1449) Paper No(s) 6/25/01; 11/4/02; 5/9/02; and 1/20/04
17) ___ Interview Summary (PTO-413) Paper No(s). _____.
18) ___ Notice of Informal Patent Application (PTO-152)
19) X Other: Text Doc for USP6,338,160

DETAILED ACTION

1. Request for copy of Applicant's response on floppy disk:
Please help expedite the prosecution of this application by including, along with your amendment response in paper form, an electronic file copy in WordPerfect, Microsoft Word, or in ASCII text format on a 3½ inch IBM format floppy disk. Please include all pending claims along with your responsive remarks. Only the paper copy will be entered -- your floppy disk file will be considered a duplicate copy. Signatures are not required on the disk copy. The floppy disk copy is not mandatory, however, it will help expedite the processing of your application. Your cooperation is appreciated.

2. **Obviousness-type double patenting rejection**

The nonstatutory double patenting rejection is based on a judicially created doctrine grounded in public policy (a policy reflected in the statute) so as to prevent the unjustified or improper timewise extension of the "right to exclude" granted by a patent and to prevent possible harassment by multiple assignees. See In re Goodman, 11 F.3d 1046, 29 USPQ2d 2010 (Fed. CIT. 1993); In re Longi, 759 F.2d 887, 225 USPQ 645 (Fed. Cir. 1985); In re van Ornurn, 686 F.2d 937, 214 USPQ 761 (CCPA 1982); In re Uogel, 422 F.2d 438, 164 USPQ 619 (CCPA 1970); and In re Thorington,418 F.2d 528, 163 USPQ 644 (CCPA 1969).

Terminal Disclaimer

3. A timely filed terminal disclaimer in compliance with 37 C.F.R. ' 1.321(b) would overcome an actual or provisional rejection on this ground provided the conflicting application or patent is shown to be commonly owned with this application. See 37 C.F.R. ' 1.78(d).

Effective January 1, 1994, a registered attorney or agent of record may sign a terminal disclaimer. A terminal disclaimer signed by the assignee must fully comply with 37 CFR 3.73(b).

4. Claims 1-15 are provisionally rejected under the judicially created doctrine of obviousness-type double patenting as being unpatentable over claims 1-30 of copending Application No. 09/703,449. Although the conflicting claims are not identical, they are not patentably distinct from each other because of corresponding language that recites many of the same elements and functions. *For example claim 1 of the instant Application claims:*

A method of creating data structures suitable for use by a virtual machine to execute a Java Load Constant instruction, the method comprising:
converting one or more Java bytecodes representing a Java Load Constant instruction in a single stream, to a representation of the Java Load Constant command in a pair of Java bytecode streams, the pair of Java bytecode streams including:
a Java code stream having one or more Java bytecodes representing the Java Load Constant command,
and a Java data stream with one or more Java bytecodes representing data associated with the Java Load Constant command in the Java code stream.

As opposed to claim 21 of Application 09/703,449
A method of creating data structures suitable for use by a virtual machine to execute Java instructions, the method comprising:
converting a java compliant bytecode into a pair of java bytecode streams having a Java code stream that includes Java commands and a Java data stream that includes the data associated with the commands included in the code stream

5. Claim Rejections - 35 U.S.C. § 101

The following is a quotation of the appropriate paragraphs of 35 U.S.C. § 101 that form the basis for the rejections under this section made in this Office action:

> "Whoever invents or discovers any new and useful process, machine, manufacture, or composition of matter or any new and useful improvement thereof, may obtain a patent therefore, subject to the conditions and requirements of this title".

6. Claims 7-10 are rejected under 35 U.S.C. § 101 because they are not directed to one of the statutory classes of patentable subject matter set forth above.

The data structure recited in claim 7 is not fixed in a tangible medium, and therefore, its functionality is not realized, as required for statutory subject matter. Claims 8-10 are also rejected on 101 grounds because each of these claims depend from claim 7 and thus suffer from the same condition.

7. Claim Rejections - 35 U.S.C. § 112

The following is a quotation of the second paragraph of 35 U.S.C. 112:

The specification shall conclude with one or more claims particularly pointing out and distinctly claiming the subject matter which the applicant regards as his invention.

8. Claim 11 is rejected under 35 U.S.C. § 112, 2nd paragraph.

Claim 11 concludes with the phrase "after the associated has been fetched." This phrase lacks the object of association. This omission makes the claim vague and indefinite.

9. The U.S. Patents used in the art rejections below have been provided as text documents which correspond to the U.S. Patents. The relevant portions of the text documents are cited according to page and line numbers in the art rejections below. For the convenience of Applicant, the cited sections are highlighted in the *text documents*. Consistent with Office procedure, the U.S. Patents corresponding to the *text documents* are also included with this action.

10. Claim Rejections - 35 U.S.C. § 102

11. The following is a quotation of the appropriate paragraphs of 35 U.S.C. § 102 that form the basis for the rejections under this section made in this Office action:

A person shall be entitled to a patent unless

(e) the invention was described in a patent granted on an application for patent by another filed in the United States before the invention thereof by the applicant for patent, or on an international application by another who has fulfilled the requirements of paragraphs (1), (2), and (4) of section 371(c) of this title before the invention thereof by the applicant for patent.

12. Claims 1-2, 5-7 and 11-13 are rejected under 35 U.S.C. § 102(e) as being anticipated by Patel et al. (U.S. Patent 6,338,160).

As to claim 7, Patel teaches in an object oriented programming environment (implementation of Java, abstract) a data structure for containing a load constant computer executable command (bytecode 150a ... instruction, p8 20-39) and data associated with the load constant computer executable command (data resolution field, Id.) the data structure suitable for use by a virtual machine and comprising:
a code portion having a load constant computer executable command (code for ... the normal invoke operations, Id.) and
a data stream (string 'ABC', Id.) having data corresponding to the load constant computer executable command (the invoke instruction has as one of its arguments ... the data 'ABC', Id.).

As to claim 1, Patel teaches a method of creating data structures suitable for use by a virtual machine to execute a Java Load Constant instruction (invoke 150a is run, the system ... uses the data in the data resolution field, p8 20-39) the method comprising:
converting one or more Java bytecodes representing a Java Load Constant instruction in a single stream (modification of the constant pool, Id.) to a representation of the Java Load Constant command in a pair of Java bytecode streams (resolution field and indication field, Id.) the pair of Java bytecode streams including:
a Java code stream having one or more Java bytecodes representing the Java Load Constant command (code for ... the normal invoke operations, Id.) and
a Java data stream (string 'ABC', Id.) with one or more Java bytecodes representing data associated with the Java Load Constant command in the Java code stream (the invoke instruction has as one of its arguments ... the data 'ABC', Id.).

As to claim 2, Patel (p8 20-39) teaches constructing a Constant Pool index into a Constant Pool associated with the Java Load Constant command (index which points to entry 152a of constant pool 152)
reading the appropriate structures of the Constant Pool index based on the constructed Constant Pool index (resolve instructions)
determining the corresponding Constant Value (data resolution field 160) and writing a representation of the determined constant value into one or more Java bytecodes of a stream of Java bytecodes (address of the loaded object 174 is placed into field 162 of constant pool entry 152).

As to claim 5 Patel (p8 46-52) teaches the resolution field "10" represents a numeric constant value.

As to claim 6, Patel's (ppd6-8) iscussion of Java and its intrinsic characteristics provide bytecodes that are one or more bytes.

As to claim 11, Patel teaches in an object oriented programming environment (implementation of Java, abstract) a method of executing load constant computer instructions on a virtual machine (bytecode 150a ... instruction, p8 20-39) the method comprising:
fetching a load constant command from a stream (bytecode 150a that references the constant pool, Id.)
fetching from another stream data associated with the load constant command (invoke instruction has as one of its arguments ... the data 'ABC', Id.) and executing the load constant command with the associated data after the associated has been fetched (system obtains a string 'ABC' and uses the data in the data resolution field as an index to the jump table ... and the normal invoke operations, Id.).

As to claim 12, Patel (p8 20-52) teaches the bytecode 150a is a Java load constant command.

As to claim 13, note the discussion of claim 6 supra.

13. Claim Rejections - 35 U.S.C. § 103

The following is a quotation of 35 U.S.C. 103(a) which forms the basis for all obviousness rejections set forth in this Office action:

(a) A patent may not be obtained though the invention is not identically disclosed or described as set forth in section 102 of this title, if the differences between the subject matter sought to be patented and the prior art are such that the subject matter as a whole would have been obvious at the time the invention was made

to a person having ordinary skill in the art to which said subject matter pertains. Patentability shall not be negatived by the manner in which the invention was made.

14. Claims 3-4 and 14-15 are rejected under 35 U.S.C. § 103(a) as being unpatentable over Patel as applied to claims 1 and 11 respectively, and further in view of the Admitted Prior Art (APA) from the Application background.

As to claim 3, Patel teaches the Java bytecode conversion, however, Patel does not explicitly disclose the first and second bytecodes determining the index into the constant pool.

The APA (Fig. 1B) teaches "bytecodes 156 and 157 ... represent the first and second bytes of an index into the constant pool", which corresponds to the fetching a first bytecode of a Constant Pool index associated with the Java Load Constant command;
fetching a second bytecode of the Constant Pool index associated with the Java Load Constant command; and
wherein said constructing of the Constant Pool index into a Constant Pool operates to determine an index based on said fetching of the first and the second
bytecodes.

It would have been obvious to combine the APA's teachings with Patel's system because the multiple byte index increases the indexing range, and effectively facilitates faster/more access.

As to claim 4, Patel (p2 34-47) teaches "quick variants of bytecodes" that are used to replace other bytecodes, which correspond to writing a representation of the Java Load Constant command into one or more Java bytecodes of a stream of another Java bytecodes.

As to claim 14, the APA teaches "Java bytecodes 152, 154 and 156 collectively represent a Java 'iconst' instruction . . . then an index to the constant pool is constructed", which corresponds to the determining how many bytecodes represent data associated with the Java Load Constant command. From this, one skilled in the art would have included the respective pointer incrementing for the code and data streams, because the decode functions must properly progress through each operator/operand as the JVM interprets the program, thus the pointers are fundamental in maintaining/managing the appropriate amount of code/data accessed in connection with the Java instruction.

As to claim 15, Patel (p7 11-15) teaches the "iload_n ... pushes the top local variable ... onto the stack", and it would have been obvious modification for one skilled in the art to include a Java Load Constant command operates to push a constant value on a stack, because the subject constant would then be available for processing.

15. Claims 8-10 are rejected under 35 U.S.C. § 103(a) as being unpatentable over Patel as applied to claim 7 above.

As to claim 8, Patel (p8 3-19) teaches the "bytecode does not reference a constant pool, other steps 122 are done". From this, it is obvious that certain code is not associated with the constant pool, and thus it would naturally follow to include a condition that a code portion does not include any data associated with the load constant computer executable command and the data portion does not include a load constant computer executable command, because it would have been inefficient to maintain unreferenced data.

As to claims 9-10, Patel (p6 1-7) teaches that "multiple bytecodes can be converted into a lesser number of native instructions", which suggests the Java commands comprising one or more bytecodes and each bytecode can be one or more bytes. It would have been obvious to stipulate that commands can be constructed from multiple bytecode and bytes because this provides the capability for stringing together bytecodes to build more Java commands.

16. The prior art of record and not relied upon is considered pertinent to the applicant's disclosure. Each reference disclosed below is relevant to one or more of the Applicant's claimed invention.
U.S. Patent No. 6,738,977 to Berry et al. which teaches the code and data sharing between multiple virtual machines;
U.S. Patent No. 6,330,907 to Johnson et al. which teaches the constant pool object sharing for storage optimization;
U.S. Patent No. 6,163,780 to Ross et al. which teaches the code and data replacement with condensed references;
U.S. Patent No. 6,081,665 to Nilsen et al. which teaches the substitutions of certain instructions for virtual machine efficiency; and,
U.S. Patent No. 5,815,718 to Tock which teaches the constant pool and the fundamentals for interpreting commands -- resolving references.

17. Contact Information:

Information regarding the status of an application may be obtained from the Patent Application Information Retrieval (PAIR) system.

Status information for published applications may be obtained from either Private-PAIR or Public-PAIR.

Status information for unpublished applications is available through Private-PAIR only.

For more information about the PAIR system, see http://pair-direct.uspto.gov.

Should you have questions on access to the Private PAIR system, contact the Electronic Business Center (EBC) at 866-217-9197 (toll-free).

☐ All responses sent by U.S. Mail should be mailed to:

Commissioner for Patents
PO Box 1450
Alexandria, VA 22313-1450

☐ Hand-delivered responses should be brought to Crystal Park Two, 2021 Crystal Drive, Arlington, VA., Sixth Floor (Receptionist). All hand-delivered responses will be handled and entered by the docketing personnel. Please do not hand deliver responses directly to the Examiner.

The fax phone number for the organization where this application or proceeding is assigned is 703-872-9306.

All OFFICIAL faxes will be handled and entered by the docketing personnel. The date of entry will correspond to the actual FAX reception date unless that date is a Saturday, Sunday, or a Federal Holiday within the District of Columbia, in which case the official date of receipt will be the next business day. The application file will be promptly forwarded to the Examiner unless the application file must be sent to another area of the Office, e.g., Finance Division for fee charging, etc.

☐ Any inquiry of a general nature or relating to the status of this application should be directed to the Group receptionist at **(703) 305-9600.**

□ Any inquiry concerning this communication or earlier communications from the examiner should be directed to George Opie at (703) 308-9120 or via e-mail at *George.Opie@uspto.gov*. Internet e-mail should not be used where sensitive data will be exchanged or where there exists a possibility that sensitive data could be identified unless there is an express waiver of the confidentiality requirements under 35 U.S.C. 122 by the Applicant. Sensitive data includes confidential information related to patent applications.

ZARNI MAUNG
PRIMARY EXAMINER

TITLE: Constant pool reference resolution method

INVENTOR(S): Patel, Mukesh K., Fremont, CA, United States
Dasgupta, Chitrabhanu, Mountain View, CA, United States
PATENT ASSIGNEE(S): Nazomi Communications, Inc., Santa Clara, CA, United States (U.S. corporation)

NUMBER	KIND	DATE	
PATENT INFORMATION:	US 6338160	B1	20020108 <--
APPLICATION INFO.:	US 2000-488186		20000120 (9)

RELATED APPLN. INFO.: Continuation-in-part of Ser. No. US 1998-208741, filed on 8 Dec 1998
DOCUMENT TYPE: Utility
FILE SEGMENT: GRANTED

	NUMBER	DATE	CLASS	INVENTOR
REFERENCED PATENT:	US 5265206	Nov 1993	709/316.000	Shakelford et al.
	US 5307492	Apr 1994	717/007.000	Benson
	US 5313614	May 1994	395/500.000	Goettelmann et al.
	US 5367685	Nov 1994	717/007.000	Gosling
	US 5584026	Dec 1996	707/001.000	Knudsen et al.
	US 5740441	Apr 1998	717/004.000	Yellin et al.
	US 5903899	May 1999	707/206.000	Steele, Jr.
	US 5920720	Jul 1999	717/005.000	Toutonghi et al.
	US 5953736	Sep 1999	711/006.000	O'Connor et al.
	US 5966542	Oct 1999	717/011.000	Tock
	US 6052526	Apr 2000	717/005.000	Chatt
	US 6067577	May 2000	709/305.000	Beard
	US 6071317	Jun 2000	717/004.000	Nagel
	US 6085198	Jul 2000	707/103.000	Skinner et al.
	US 6151702	Nov 2000	717/005.000	Overturf et al.
	US 6275984	Aug 2001	717/005.000	Morita

NON-PATENT REFERENCE: L. Lemay and C. L. Perkins, "Teach Yourself JAVA 1.1 in 21 Days," second edition, pp. 621-675 (1997).

PRIMARY EXAMINER: Powell, Mark R.
ASSISTANT EXAMINER: Das, Chameli Chaudhuri
LEGAL REPRESENTATIVE: Burns Doane Swecker & Mathis LLP
NUMBER OF CLAIMS: 22
EXEMPLARY CLAIM: 1
NUMBER OF DRAWINGS: 14 Drawing Figure(s); 13 Drawing Page(s)

ABSTRACT:
An implementation of Java is disclosed in which references to the constant pool are implemented by using a Data Resolution Field within the constant pool entry. The Data Resolution Field acts as an index to a jump table to jump to resolve the reference or to perform the bytecode instruction. When the reference is resolved, the contents of the Data Resolution Field in the constant pool entry are modified so that the next time the bytecode is run, the resolution steps need not be done.

RELATED U.S. APPLICATIONS

This application is a continuation-in-part of the application "Java Virtual Machine Hardware for RISC and CISC Processors", Ser. No. 09/208,741, filed Dec. 8, 1998.

BACKGROUND OF THE INVENTION

Java.TM. is an object oriented programming language developed by Sun Microsystems. The Java language is small, simple and portable across platforms and operating systems, both at the source and at the binary level. This makes the Java programming language very popular on the Internet.

Platform independence and code compaction are the most significant advantages of Java over conventional programming languages. In conventional programming languages, the source code of a program is sent to a compiler which interprets the program into machine code or processor instructions. The processor instructions are native to the system's processor. If the code is compiled on an Intel-based system, the resulting program will only run on other Intel-based systems. If it is desired to run the program on another system, the user must go back to the original source code, obtain a compiler for the new processor, and recompile the program into the machine code specific to that other processor.

Java operates differently. The Java compiler takes a Java program and, instead of generating machine code for a particular processor, generates bytecodes. Bytecodes are instructions that look like machine code, but aren't specific to any processor. To execute a Java program, a bytecode interpreter takes the Java bytecode converts them to equivalent native processor instructions and executes the Java program. The Java byte code interpreter is one component of the Java Virtual Machine.

Having the Java programs in bytecode form means that instead of being specific to any one system, the programs can run on any platform and any operating system as long a Java Virtual Machine is available. This allows a binary bytecode file to be executable across platforms.

Most computer languages, such as C, are compiled languages. All references to objects are resolved before running of the program. Because Java is run from the Virtual Machine, it is possible to operate without having all of the references resolved. The advantage of this arrangement is that it allows for operation of the program or the program is completely downloaded from another location, such as off the Internet.

For example, a Java instruction that invokes `ABC` may be run when the Class `ABC` is not loaded into memory. The Invoke instruction must cause the reference to be resolved. This can take a considerable amount of time to resolve the reference each time that the instruction is run. Instructions that reference the constant pool, such as invoke instructions, often can have this problem.

One version of a Java Virtual Machine can reduce this problem with the use of quick variants of bytecodes. The quick variants of bytecodes are not officially part of the Java Virtual Machine specification and are invisible outside specific Java Virtual Machine implementations. When the quick optimization is turned on, each non-quick bytecode resolves the specified item in the constant pool, signals if an item in the constant pool could not be resolved for some reason, turns itself into the quick variant of itself, and then performs its intended operation. Thus, the bytecode is written over by the quick bytecode variant. The next time the code is run, the system assumes that the item in the constant pool has already been resolved, and that this resolution did not produce any errors. The system can then simply performs the intended operation on the resolved item. A discussion of such a system is described in Gosling, U.S. Pat. No. 5,367,685 incorporated herein by reference.

This optimization cannot be run from read-only memory (ROM) because the optimization requires writing over the normal bytecode with its quick variant. Running a program from read-only memory could be quite useful in some circumstances particularly for embedded systems. It is desired to have an improved method of resolving the constant pool references to avoid some of the problems of the prior art.

SUMMARY OF THE INVENTION

The present invention, a Resolution Data Field, is used in the constant pool entries. When a reference to the constant pool is done by a bytecode, the data in the resolution data field acts as an index to a jump table to determine the native code to be run next. The first time a bytecode, which references the

constant pool, is run, the data in the resolution data field causes the system to jump to code to resolve the reference. This resolution code could search for the object reference bytecode within the memory. If the object is not within the memory, the system can load the object into memory and then operate in the normal bytecode operation. This system also sets the resolution data field, so as to indicate that the reference has been resolved. An indication of the location of the resolved object is also stored within the constant pool entry. The next time the bytecode is operated on, the data in the resolution data field acts as an index to the jump table which causes a jump to operation code that assumes that the resolution has been resolved. In this way, after the first operation of the bytecode, searching for the object in memory need not be done.

The advantage of the present invention is that it is not required to write the bytecode over with another value, such as a quick bytecode. Thus, the bytecode could be run from read-only memory. The advantage of using a jump table in the preferred embodiment is that the data in the resolution data field need only be added to a base value rather than compared to a value. In most current processors, the add step can be done quicker than a compare step. In an alternate embodiment, a comparison of the data in the resolution data field could be done.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention may be further understood from the following description in conjunction with the drawings.

FIG. 1 is a diagram of the system of the parent case including the hardware Java accelerator.

FIG. 2 is a diagram illustrating the use of the hardware Java accelerator of the parent case.

FIG. 3 is a diagram illustrating some the details of a Java hardware accelerator of one embodiment of the parent case.

FIG. 4 is a diagram illustrating the details of one embodiment of a Java accelerator instruction translation in the system of the parent case.

FIG. 5 is a diagram illustration the instruction translation operation of one embodiment of the parent case.

FIG. 6 is a diagram illustrating the instruction translation system of one embodiment of the parent case using instruction level parallelism.

FIG. 7 is a table showing one possible list of bytecodes which can cause exceptions in a preferred embodiment of the parent case.

FIG. 8 is a flow chart illustrating the operation of the present invention.

FIG. 9A and 9B are diagrams illustrating the operation of one embodiment of the present invention

FIG. 10 is a diagram illustrating one embodiment of the constant pool entry for use with one embodiment of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

FIGS. 1-7 illustrate the operation of one embodiment of the application of the parent case. FIGS. 8-10 illustrate the operation of the present invention.

FIG. 1 is a diagram of the system 20 showing the use of a hardware Java accelerator 22 in conjunction with a central processing unit 26. The Java hardware accelerator 22 allows part of the Java Virtual Machine to be implemented in hardware. This hardware implementation speeds up the processing of the Java byte codes. In particular, in a preferred embodiment, the translation of the Java bytecodes into native processor instructions is at least partially done in the hardware Java accelerator 22. This translation has been part of a bottleneck in the Java Virtual Machine when implemented in software. In FIG. 1, instructions from the instruction cache 24 or other memory is supplied to the hardware Java accelerator 22. If these instruction are Java bytecode, the hardware Java accelerator 22 can convert these bytecodes into native processor instruction which are supplied through the multiplexer 28 to

the CPU. If a non-Java code is used, the hardware accelerator can be by-passed using the multiplexer 26.

The Java hardware accelerator can do, some or all of the following tasks:

1. Java bytecode decode;
2. identifying and encoding instruction level parallelism (ILP), wherever possible;
3. interpreting bytecodes to native instructions;
4. managing the Java stack on a register file associated with the CPU or as a separate stack;
5. generating exceptions on instructions on predetermined Java byte codes;
6. switching to native CPU operation when native CPU code is provided;
7. performing bounds checking on array instructions; and
8. managing the variables on the register file associated with the CPU.

In a preferred embodiment, the Java Virtual Machine functions of bytecode interpreter, Java register, and Java stack are implemented in the hardware Java accelerator. The garbage collection heap and constant pool area can be maintained in normal memory and accessed through normal memory referencing.

The major advantages of the Java hardware accelerator is to increase the speed in which the Java Virtual Machine operates, and allow existing native language legacy applications, software base, and development tools to be used. A dedicated microprocessor in which the Java bytecodes were the native instructions would not have access to those legacy applications.

Although the Java hardware accelerator is shown in FIG. 1 as separate from the central processing unit, the Java hardware accelerator can be incorporated into a central processing unit. In that case, the central processing unit has a Java hardware accelerator subunit to interpret Java bytecode into the native instructions operated on by the main portion of the CPU.

FIG. 2 is a state machine diagram that shows the operation of one embodiment of the parent case. Block 32 is the power-on state. During power-on, the multiplexer 28 is set to bypass the Java hardware accelerator. In block 34, the native instruction boot-up sequence is run. Block 36 shows the system in the native mode executing native instructions and by-passing the Java hardware accelerator.

In block 38, the system switches to the Java hardware accelerator mode. In the Java hardware accelerator mode, Java bytecode is transferred to the Java hardware accelerator 22, converted into native instructions then sent to the CPU for operation.

The Java accelerator mode can produce exceptions at certain Java bytecodes. These bytecodes are not processed by the hardware accelerator 22 but are processed in the CPU 26. As shown in block 40, the system operates in the native mode but the Java Virtual Machine is implemented in the CPU which does the bytecode translation and handles the exception created in the Java accelerator mode.

The longer and more complicated bytecodes that are difficult to handle in hardware can be selected to produce the exceptions. FIG. 7 is a table showing one possible list of bytecodes which can cause exceptions in a preferred embodiment. An implementation of bytecode instruction operations that reference the constant pool is discussed below with respect to FIGS. 8-10.

FIG. 3 is a diagram illustrating details of one embodiment of the Java hardware accelerator of the parent case. The Java hardware accelerator includes Java accelerator instruction translation hardware 42. One embodiment of the Java accelerator instruction translation hardware 42 is described in more detail below with respect to FIG. 4. This instruction translation hardware 42 uses data stored in hardware Java registers 44. The hardware Java Registers store the Java Registers defined in the Java Virtual Machine. The Java Registers contain the state of the Java Virtual Machine, affect its operation, and are updated after each bytecode is executed. The Java registers in the Java virtual

machine include the PC, the program counter indicating what bytecode is being executed; Optop, a pointer to the top of the operand stack; Frame, a pointer to the execution environment of the current method; and Vars, a pointer to the first local variable available of the currently executing method. The virtual machine defines these registers to be a single 32-bit word wide. The Java registers are also stored in the Java stack which can be implemented as the hardware Java stack 50 or the Java stack can be stored into the CPU associated register file.

In a preferred embodiment, the hardware Java registers 44 can include additional registers for the use of the instruction translation hardware 42. These registers can include a register indicating a switch to native instructions and a register indicating the version number of the system.

The Java PC can be used to obtain bytecode instructions from the instruction cache 24. In one embodiment the Java PC is multiplexed with the normal program counter 54 of the central processing unit 26 in multiplexer 52. The normal PC 54 is not used during the operation of the Java hardware bytecode translation. In another embodiment, the normal program counter 54 is used as the Java program counter.

The Java registers are a part of the Java Virtual Machine and should not be confused with the general registers 46 or 48 which are operated upon by the central processing unit 26. In one embodiment, the system uses the traditional CPU register file 46 as well as a Java CPU register file 48. When native code is being operated upon the multiplexer 56 connects the conventional register file 46 to the execution logic 26c of the CPU 26. When the Java hardware accelerator is active, the Java CPU register file 48 substitutes for the conventional CPU register file 46. In another embodiment, the conventional CPU register file 46 is used.

As described below with respect to FIGS. 3 and 4, the Java CPU register file 48, or in an alternate embodiment the conventional CPU register file 46, can be used to store portions of the operand stack and some of the variables. In this way, the native register-based instructions from the Java accelerator instruction translator 42 can operate upon the operand stack and variable values stored in the Java CPU register file 48, or the values stored in the conventional CPU register file 46. Data can be written in and out of the Java CPU register file 48 from the data cache or other memory 58 through the overflow/underflow line 60 connected to the memory arbiter 62. The overflow/underflow transfer of data to and from the memory to can done concurrently with the CPU operation. Alternately, the overflow/underflow transfer can be done explicitly while the CPU is not operating. The overflow/underflow bus 60 can be implemented as a tri-state bus or as two separate buses to read data in and write data out of the register file when the Java stack overflows or underflows.

The register files for the CPU could alternately be implemented as a single register file with native instructions used to manipulate the loading of operand stack and variable values to and from memory. Alternately, multiple Java CPU register files could be used: one register file for variable values, another register file for the operand stack values, and another register file for the Java frame stack holding the method environment information.

The Java accelerator controller (co-processing unit) 64 can be used to control the hardware Java accelerator, read in and out from the hardware Java registers 44 and Java stack 50, and flush the Java accelerator instruction translation pipeline upon a "branch taken" signal from the CPU execute logic 26c.

The CPU 26 is divided into pipeline stages including the instruction fetch 26a, instruction decode 26b, execute logic 26c, memory access logic 26d, and writeback logic 26e. The execute logic 26c executes the native instructions and thus can determine whether a branch instruction is taken and issue the "branch taken" signal.

FIG. 4 illustrates an embodiment of a Java accelerator instruction translator which can be used with the parent case. The instruction buffer 70 stores the bytecode instructions from the instruction cache. The bytecodes are sent to a parallel decode unit 72 which decodes multiple bytecodes at the same time. Multiple bytecodes are processed concurrently in order to allow for instruction level parallelism. That is, multiple bytecodes may be converted into a lesser number of native instructions.

The decoded bytecodes are sent to a state machine unit 74 and Arithmetic Logic Unit (ALU) 76. The ALU 76 is provided to rearrange the bytecode instructions to make them easier to be operated on by the state machine 74. The state machine 74 converts the bytecodes into native instructions using the look-up table 78. Thus, the state machine 74 provides an address which indicates the location of the desired native instruction in the look-up table 78 . Counters are maintained to keep a count of how many entries have been placed on the operand stack, as well as to keep track of the top of the operand stack. In a preferred embodiment, the output of the look-up table 78 is augmented with indications of the registers to be operated on at line 80. The register indications are from the counters and interpreted from bytecodes. Alternately, these register indications can be sent directly to the Java CPU register file 48 shown in FIG. 3.

The state machine 74 has access to the Java registers in 44 as well as an indication of the arrangement of the stack and variables in the Java CPU register file 48 or in the conventional CPU register file 46. The buffer 82 supplies the interpreted native instructions to the CPU.

The operation of the Java hardware accelerator of one embodiment of the parent case is illustrated in FIGS. 5 and 6. FIG. 5, section I shows the instruction translation of the Java bytecode. The Java bytecode corresponding to the mnemonic iadd is interpreted by the Java virtual machine as an integer operation taking the top two values of the operand stack, adding them together and pushing the result on top of the operand stack. The Java translating machine interprets the Java bytecode into a native instruction such as the instruction ADD R1, R2. This is an instruction native to the CPU indicating the adding of value in register R1 to the value in register R2 and the storing of this result in register R2. R1 and R2 are the top two entries in the operand stack.

As shown in FIG. 5, section II, the Java register includes a PC value of "Value A" that is incremented to "Value A+1". The Optop value changes from "Value B" to "Value B-1" to indicate that the top of the operand stack is at a new location. The Vars value which points to the top of the variable list is not modified. In FIG. 5, section III, the contents of a Java CPU register file, such as the Java CPU register file 48 in FIG. 3, is shown. The Java CPU register file starts off with registers R0-R5 containing operand stack values and registers R6-R7 containing variable values. Before the operation of the native instruction, register R1 contains the top value of the operand stack. Register R6 contains the first variable. After the execution of the native instruction, register R2 now contains the top value of the operand stack. Register R1 no longer contains a valid operand stack value and is available to be overwritten by a operand stack value from the memory sent across the overflow/underflow line 60 or from the bytecode stream.

FIG. 5, section IV, shows the memory locations of the operand stack and variables which can be stored in the data cache 58 or in main memory. For convenience, the memory is illustrated without illustrating any virtual memory scheme. Before the native instruction executes, the address of the top of the operand stack, Optop, is "Value B". After the native instruction executes, the address of the top of the operand stack is "Value B-1" containing the result of the native instruction. Note that the operand stack value "4427" can be written into register R1 across the overflow/underflow line 60. Upon a switch back to

the native mode, the data in the Java CPU register file 48 should be written to the data memory.

Consistency must be maintained between the Hardware Java Registers 44, the Java CPU register file 48 and the data memory. The CPU 26 and Java Accelerator Instruction Translation Unit 42 are pipelined and any changes to the hardware java registers 44 and changes to the control information for the Java CPU register file 48 must be able to be undone upon a "branch taken" signal. The system preferably uses buffers (not shown) to ensure this consistency. Additionally, the Java instruction translation must be done so as to avoid pipeline hazards in the instruction translation unit and CPU.

FIG. 6 is a diagram illustrating the operation of instruction level parallelism with the parent case. In FIG. 6 the Java bytecodes iload_n and iadd are converted by the Java bytecode translator to the single native instruction ADD R6, R1. In the Java Virtual Machine, iload_n pushes the top local variable indicated by the by the Java register VAR onto the operand stack.

In the parent case the Java hardware translator can combine the iload_n and iadd bytecode into a single native instruction. As shown in FIG. 6, section II, the Java Register, PC, is updated from "Value A" to "Value A+2". The Optop value remains "value B". The value Var remains at "value C".

As shown in FIG. 6, section III, after the native instruction ADD R6, R1 executes the value of the first local variable stored in register R6, "1221", is added to the value of the top of the operand stack contained in register R1 and the result stored in register R1. In FIG. 6, section IV, the Optop value does not change but the value in the top of the register contains the result of the ADD instruction, 1371.

The Java hardware accelerator of the parent case is particularly well suited to a embedded solution in which the hardware accelerator is positioned on the same chip as the existing CPU design. This allows the prior existing software base and development tools for legacy applications to be used. In addition, the architecture of the present embodiment is scalable to fit a variety of applications ranging from smart cards to desktop solutions. This scalability is implemented in the Java accelerator instruction translation unit of FIG. 4. For example, the lookup table 78 and state machine 74 can be modified for a variety of different CPU architectures. These CPU architectures include reduced instruction set computer (RISC) architectures as well as complex instruction set computer (CISC) architectures. The parent case can also be used with superscalar CPUs or very long instruction word (VLIW) computers.

The term Java in the specification or claims should be construed to cover successor programming languages or other programming languages using basic Java concepts (the use of generic instructions, such as bytecodes, to indicate the operation of a virtual machine).

FIG. 8 is a flow chart that illustrates the apparatus of one embodiment of the present invention. In step 120, there is a check whether a bytecode references specific data structure, for example the constant pool. Every currently loaded class has a constant pool attached to it. The constant pool is allocated when a class is first loaded, the constants in this pool encode all the names (of variables, methods, and so forth) used by any method in the class. The class contains a count of how many constants there are and the start of the constant pool is available to the Java bytecodes. The bytecodes that reference the constant pool and include load constant instructions (ldc, ldc_w, ldc2_w), anewarray allocation, multinewarray allocation, getfield operations, putfield operations, getstatic operations, putstatic operations, the invoke operations (invoke interface operations, invoke special operations, invoke static operations and invoke virtual operations), new operation, checkcast operations, and instanceof operations.

Step 120 is preferably done in hardware as described with respect to FIGS. 1-7 above. The references to the constant pool are software exceptions for that

hardware implementation. Alternately, the present invention of FIGS. 8-10 can be completely implemented in software.

If the bytecode does not reference a constant pool, other steps 122 are done. If the bytecode does reference a constant pool, in step 124, data is obtained from the specific data structure, i.e. the constant pool, including data from a resolution data field. The data from the resolution data field is used as an index to a jump table. In steps 126 and 128, the operation of the jump table checks whether the reference is resolved. If the reference is resolved, the system jumps to do the normal operation on the resolved reference in block 130. If the reference is not resolved, the system jumps to the resolution operation to check whether the object referred to is in memory. If the object referred to is not in memory, the object is loaded into memory in step 132. The data in the data structure is modified to indicate in step 134 that the reference is resolved. Thus, the constant pool entry will indicate the location of the found or loaded reference. The resolution data field is also set, so as to indicate that the reference is resolved. In step 136, the bytecode is done upon the resolved reference. Step 138 indicates getting the next bytecode. Note that the system moves to steps 126 and 128 since the resolution data field is set. The resolution data field will indicate that the reference is resolved.

Details of one embodiment of the present invention is shown with respect to FIG. 9A and 9B. Looking at FIG. 9A, different elements stored in the memory are shown. In the bytecode region 150, includes a bytecode 150a that references the constant pool 152. The operation of the invoke instruction has as one of its arguments, the index which points to entry 152a in the constant pool 152. The constant pool entries include a resolution data field 160 and indication field 162. In this case, the indication `5002` points to address `5002` that contains the data `ABC`. The resolution data field 160 indicates that the reference to the object has not been resolved. The first time that the instruction invoke 150a is run, the system obtains a string `ABC` and uses the data in the data resolution field 160 as an index to the jump table 170. The `0` data in the data resolution field 160 causes the system to jump to location `9000` in the native instruction region 172. At address `9000`, the resolve instructions include code for the object search, the loading of the object if not in memory, the modification of the constant pool, and the normal invoke operations. FIG. 9B illustrates the operation of the native instruction starting at address `9000`. The object `ABC` was not found in memory so is loaded starting at location `7500`. The address of the loaded object 174 is placed into the in the field 162' of constant pool entry 152a . The data in the data resolution field 160' is modified to `1`. The invoke bytecode 150a is not modified.

The next time the invoke bytecode 150a is run, the data in the data resolution field 160' will cause a jump in the jump table 170 to location `9500`. This will be the native instruction for the invoke instruction. The contents of the field 162' indicates the location of the loaded object 174. Note that there can be different jump tables and instruction regions for each specific bytecode instruction referencing the constant pool.

FIG. 10 illustrates one embodiment of a constant pool entry 180. The constant pool entry 180 includes a field 180b for the Java object address or nonJava address, also shown as the reference resolution field 180a, The reference resolution field can also indicate whether the reference is to a Java object. In one embodiment, "00" indicates an unresolved Java object, "01" indicates a resolved Java object, "10" indicates a numeric constant, and "11" indicates a text string. Field 180c contains garbage collection bits.

While the present invention has been described with reference to the above embodiments, this description of the preferred embodiments and methods is not meant to be construed in a limiting sense. It should also be understood that all aspects of the present invention are not to be limited to the specific descriptions, or to configurations set forth herein. Variations in the present

invention will be apparent to a person skilled in the art upon reference to the present disclosure. It is therefore contemplated that the following claims will cover any such modifications or variations of the described embodiment as falling within the true spirit and scope of the present invention.

What is claimed is:

1. A method of executing an instruction comprising: obtaining from an instruction storage location, an instruction that references a data structure, the data structure storing an indication of a reference that may need resolution; obtaining data from the data structure including data from a resolution data field; using data from resolution data field as an index to a jump table to determine whether to do a resolving step; and thereafter, if the data in the data resolution field indicates that the reference was not resolved, resolving the reference and, thereafter, modifying the data in the data structure including modifying the data in the resolution data field to indicate that the reference is resolved, wherein the data in the instruction storage location is not modified.

2. The method of claim 1, wherein the instruction comprises a bytecode.

3. The method of claim 1, wherein the data structure is a constant pool entry.

4. The method of claim 3, wherein the instruction includes an indexed reference to the constant pool entry.

5. The method of claim 4, wherein the reference is an invoke reference.

6. The method of claim 1, wherein the indication of a reference points to a label of an object that may or may not be loaded into memory.

7. The method of claim 1, wherein the resolving step includes searching for an object in memory.

8. The method of claim 7, further comprising loading the object into memory.

9. The method of claim 8, further comprising storing, in the data structure, an indication of a memory location that the object is stored into memory.

10. The method of claim 1, wherein the jump table contains addresses of regions of native code in memory.

11. A method of executing an instruction comprising: obtaining from an instruction storage location, an instruction that references an entry in a constant pool, the constant pool entry storing an indication of a reference that may need resolution; obtaining data from the constant pool entry including data from a resolution data field; using data from the resolution data field to determine whether to do a resolving step; and thereafter, if the data in the data resolution field indicates that the reference was not resolved, resolving the reference and, thereafter, modifying the data in the constant pool entry including modifying the data in the resolution data field to indicate that the reference is resolved, wherein the data in the instruction storage location is not modified.

12. The method of claim 11, wherein data from the resolution data field is used as an index to a jump table to determine whether to do a resolving step.

13. The method of claim 12, wherein the jump table contains addresses of regions of native code in memory.

14. The method of claim 11, wherein the instruction comprises a bytecode.

15. The method of claim 11, wherein the instruction includes an indexed reference to the constant pool entry.

16. The method of claim 15, wherein the reference is an invoke reference.

17. The method of claim 11, wherein the indication of a reference points to a label of an object that may or may not be loaded into memory.

18. The method of claim 11, wherein the resolving step includes searching for an object in memory.

19. The method of claim 18, further comprising loading the object into memory.

20. The method of claim 18, further comprising storing, in the constant pool entry, an indication of a memory location that the object is stored into memory.

21. The method of claim 11, further comprising the step of examining instructions to determine which instructions reference the constant pool.
22. The method of claim 21, wherein the examining step is implemented in hardware.

ISSUE U.S. PATENT CLASSIF.:
MAIN: 717/005.000
SECONDARY: 717/004.000; 717/007.000; 717/011.000; 709/316.000
CURRENT U.S. PATENT CLASSIF.:
MAIN: 717/139.000
SECONDARY: 717/118.000; 717/148.000; 719/316.000
INT. PATENT CLASSIF.: [7]
MAIN: G06F009-15
FIELD OF SEARCH: 717/5; 717/4; 717/7; 717/11; 717/316
ART UNIT: 212

```
(FILE 'HOME' ENTERED AT 11:17:58 ON 28 JUN 2004)
     FILE 'USPATFULL' ENTERED AT 11:18:14 ON 28 JUN 2004
                SET HIGH OFF
L1           139 S 717/120-121/NCL NOT AD>20001031
L2           189 S 717/140/NCL NOT AD>20001031
L3           315 S 717/146-148/NCL NOT AD>20001031
L4           295 S 717/151-153/NCL NOT AD>20001031
L5           172 S 717/159/NCL NOT AD>20001031
L6           363 S 717/162-166/NCL NOT AD>20001031
L7           150 S 718/1/NCL NOT AD>20001031
L8           655 S 718/100/NCL NOT AD>20001031
L9          1568 S 719/310-320/NCL NOT AD>20001031
L10         1568 S 719/310-320/NCL NOT AD>20001031
L11         8701 S (717/?/NCL OR 718/?/NCL OR 719/?/NCL) NOT AD>20001031
                SET HIGH ON
L12        22096 S JAVA OR JVM
L13          661 S QUICKENING
L14            2 S QUICKENING (P) (BYTECODE OR BYTE CODE OR JAVA OR JVM)
L15            0 S L11 AND L13
L16          145 S (JAVA OR BYTECODE OR BYTE CODE) (10A) (LOAD? OR INVOK? OR INV
L17          145 FOCUS L16 1-
L18           85 S (JAVA OR BYTECODE OR BYTE CODE) (15A) (INSTRUCTION? OR COMMAN
L19           21 S L11 AND L18
L20          846 S CONSTANT (5A) POOL
L21          211 S (JAVA OR JVM OR BYTECODE OR BYTE CODE) (P) (CONSTANT (5A) POO
L22          211 FOCUS L21 1-
L23           11 S 1-3 AND L21
L24           19 S L4-6 AND L21
L25            9 S L7 AND L21
L26            2 S L8 AND L21
L27            8 S L9 AND L21
L28            0 S L10 AND L21 NOT L23-27
L29         1448 S (INSTRUCTION OR COMMAND OR CODE OR BYTECODE OR BYTE CODE) (10
L30           37 S L12 (P) L29
L31          101 S L12 (L) L29
L32           64 S L1-9 AND L29 NOT L23-27
L33           34 S L20 (L) L29
L34          303 S (CONVER? OR REPLAC? OR SUBSTITUT?) (10A) (BYTECODE OR BYTE CO
L35            0 S (CONVER? OR REPLAC? OR SUBSTITUT?) (10A) (BYTECODE OR BYTE CO
L36          303 FOCUS L34 1-
L37          632 S (CONVER? OR REPLAC? OR SUBSTITUT?) (10A) (BYTECODE OR BYTE CO
L38           89 S L29 (L) L37
L39           31 S L12 (L) L37
L40          920 S LOAD? (5A) CLASS (P) (BYTECODE OR BYTE CODE OR JAVA OR JVM)
L41            1 S L37 (P) L40
L42           46 S (LOAD? (5A) CLASS (P) (BYTECODE OR BYTE CODE OR JAVA OR JVM))
L43            3 S L37 (L) L40
L44           84 S L20 (3P) L40
L45           40 S L11 AND L44
                E SOKOLOV ?/IN
L46           36 S E78
                E WALLMAN D?/IN
L47           27 S E4
L48            1 S US6738977/PN
L49            1 S US6654778/PN
L50            1 S US6385764/PN
L51            1 S US6338160/PN
```

Case 09/703,356
IMPROVED METHODS AND APPARATUS FOR NUMERIC CONSTANT VALUE INLINING IN VIRTUAL MACHINES

1 July 2004, consultation with Jack Harvey on 101 issue.